Onderzoeksrapport Nr. 8006

A DYNAMIC PROGRAMMING ALGORITHM FOR DISPLAYING DECISION TABLES

by

J. VANTHIENEN

Wettelijk Depot : D/1980/2376/15

Katholieke Universiteit Leuven, Department of Applied Economic Sciences

This work was supported by the C.I.M. , project no. 10.

CONTENTS

0. INTRODUCTION

In computer based procedural decision modeling, the conversion of constructed decision tables from internal matrix representation to meaningfull decision table output is a common problem.

This problem is largely caused by the requirement that consecutive equal condition states should be displayed only once for reasons of clarity and space limitations, and that the table width should be kept to a minimum.

In this paper an algorithm is presented which minimizes the width (in terms of print positions) of the displayed table. It has recently been applied succesfully in the interactive procedural decision modeling system, PRODEMO (1).

---

(1) For more information on PRODEMO, see :
MAES R., VANTHIENEN J. : PRODEMO : PROcedural DEcision MOdeling through the use of decision tables, Bedrijfseconomische Verhandeling nr. 8002, K.U.Leuven, Department of Applied Economic Sciences, 1980, 34 p.

1. PROBLEM ILLUSTRATION

As an example, we will use the decision table in fig. 1, which might e.g. originate from the following text :

> "A discount of 5 % is given for normal editions and hard covers, except for wholesalers who order more than 100 hard cover books and get then 10 %, and except for orders consisting of less than (or equal to) 100 normal editions.
> No customer gets a discount for pockets or for less than 100 normal editions."

The table (in contracted form) looks like this : (1)

BOOKSHOP DISCOUNT TABLE

Type of book	hard cover			normal edition		pocket
Type of customer	wholesaler		retailer	-		-
Quantity ordered	up to 100	more than 100	-	up to 100	more than 100	-
No discount	-	-	-	x	-	x
5 % discount	x	-	x	-	x	-
10 % discount	-	x	-	-	-	-

Fig. 1 : Example table

This is the final table layout. Let us now see how this table can be displayed.

---

(1) We will not be dealing here with methods for constructing decision tables (either manual or automatic). We assume the table has been constructed correctly, checked for inconsistencies, incompleteness and contradictions, ...

The decision table in fig. 1 consists of the following elements :

conditions		condition states
type of book	:	hard cover, normal edition, pocket
type of customer	:	wholesaler, retailer
quantity ordered	:	up to 100, more than 100

actions

no discount
5 % discount
10 % discount

When a computer based system, like PRODEMO, is used to construct the table, the names of conditions, states and actions are represented by their number to facilitate the manipulation during the construction process. The decision table is then represented internally by matrix tk(k,i) for the condition part and matrix ta(a,i) for the action part, as indicated in fig. 2.

decision columns

condition	1	1	1	1	2	2	3	
	2	1	1	2	0	0	0	tk(k,i)
	3	1	2	0	1	2	0	
action	1	0	0	0	1	0	1	
	2	1	0	1	0	1	0	ta(a,i)
	3	0	1	0	0	0	0	

Fig. 2 : Internal table representation

The elements of matrix tk(k,i) indicate the state number of condition k in decision column i. A '0' in tk(k,i) means that condition k is irrelevant in decision rule i (indicated by '-' in the decision table).

A '1' in matrix ta(a,i) denotes that action a should be executed in decision rule i. The elements '1' and '0' of ta(a,i) correspond with action entries 'x' and '-' respectively.

The display problem consists of transforming the internal table representation (fig. 2) to the standard decision table format (fig. 1). Attention has to be paid however to the requirement that subsequent equal condition states should be displayed only once in order to minimize the table width.

We therefore need the following information :

k^{max}	: number of conditions (=3)
a^{max}	: number of actions (=3)
$s^{max}(k)$	: number of states of condition k (=3,2,2)
n	: number of decision columns (=6)
tk(k,i)	: state number of condition k in column i
ta(a,i)	: action indicator of action a in column i
sl(k,s)	: length of state s of condition k (1)

with : $k = 1, \ldots, k^{max}$

$a = 1, \ldots, a^{max}$

$s = 0, \ldots, s^{max}(k)$ and sl(k,0) = 1 (i.e. '-')

$i = 1, \ldots, n$

All these elements are supposed to be known at the start of the display procedure.

---

(1) For various reasons PRODEMO only allows limited entry actions ('x' or '-'). We will therefore not take into account action entry length, however simple it might be.

The length of the states is stored in matrix sl(k,s), which also accounts for accents ('$,\acute{},\grave{},\ddot{},\hat{},\ldots$), capital letters and various special symbols (&,_,?,...). This length is expressed in terms of occupied screen positions (1).

Displaying the decision table can be split up into (at least) two parts :

1. Layout and display of the stub (i.e. the condition and action names)
2. Determination of the column boundaries and displaying the decision columns.

The main problem is in the last step : the determination of the rule boundaries, for which a dynamic programming algorithm has to be developped.

---

(1) As the PRODEMO system has been implemented on the video-screen oriented PLATO system, we will talk in terms of screen positions. There is however no difficulty in translating this to printing positions.

2. DISPLAYING THE STUB

The stub consists of condition and action names. Each of them has to be displayed within a certain area, between the first screen position and the stubwidth, for ease of interpretation of the decision table.

There are two important restrictions :

- A condition or action name may consist of several words and the maximal length exceeds the stubwidth, so the names have to be split up among several lines.
- The screen is limited in the number of lines. All names however should be displayed on one single screen for reasons of clarity. (1)

According to these restrictions, there are two major problems :

- When condition or action names are split up, special symbols ('$,\acute{}$,`$,\hat{}$,¨$,\_$,...) should be displayed on the same line as the letters they belong to.
- The screen being limited, the number of lines available will sometimes be insufficient to display all condition and action names. In that case the stubwidth should be enlarged or, if this is not possible anymore, the names have to be truncated.

As there is complete analogy in displaying condition and action names, we will focus on the condition part.

---

(1) A standard PLATO terminal is equipped with a 32 x 64 screen for normal characters, each character being built of 16 x 8 dots. Each dot in this 512 x 512 address space can be accessed individually to facilitate graphics and to create visually attractive displays.

A distinction has to be made between the displayed and the real (i.e. internal) length of a character string (e.g. a condition name). The real length is the number of internal memory positions to store the string and can largely be compared with the number of keypresses. The displayed length however indicates the number of occupied screen positions.

E.g. : capital letter 'A'
real length = 2 (shift,a)
displayed length = 1

underlined letter 'a'
real length = 3 (a,backspace, _)
displayed length = 1

Bearing this in mind, we can roughly apply the following procedure :

If the displayed length (and a fortiori : if the real length) of the name is smaller than the stubwidth, the name can be displayed on one line.

e.g. : |Condition 1|

If the displayed length exceeds the stubwidth, the name has to be split among subsequent lines. A line is then assigned characters as long the displayed length is not greater than the stubwidth.

e.g. :
```
|This is con|
|dition numb|
|er two     |
```

It can be seen that the 'n' is not split among two lines, as the '_' (consisting of : backspace,_) adds nothing to the displayed length.

Ignoring the displayed length would result in :

This is c
ondition
number two

because the shift and backspace keys are now counted as separate characters of displayed length 1.

The actual PRODEMO implementation accounts for leading blanks and splits the names according to spaces between words. This small addition highly improves the stub layout.

E.g. :

This is
condition
number two

If at any moment the line considered reaches the screen bottom, the process stops and restarts with a larger stubwidth (1).

If the stubwidth is at its maximum and the number of lines still seems insufficient (which is very exceptionnal), the names are truncated.

This procedure causes no special difficulties and will therefore not be detailed further here. The only problem (in terms of execution efficiency) is the character-per-character analysis of condition and action names to be able to account for capital letters, half- and backspaces, accents and various special symbols.

The stub layout is calculated only once for a given table. As long as condition and action names remain unchanged, the results of this analysis are kept.

---

(1) Stubwidths in PRODEMO are (for another reason) fixed at 10, 20, 30 or 40.

3. DISPLAYING THE DECISION COLUMNS

The problem is split up in two parts :

- determination of the rule boundaries (the vertical lines)
- filling in the condition states in these boundaries

This subdivision is caused by the requirement of a nice left right top down display, i.e. we first calculate the positions and then print from left to right and from top to bottom.

3.1. Pass 1 : determination of the rule boundaries : the dynamic programming problem

This first part deals with the calculation of the positions where the vertical lines between the decision columns have to be drawn.

3.1.1. Methodology

Our objective is to minimize the table width, given the length of the states, the table structure and the starting position.
The starting position, the stubwidth, is called bound(0). The horizontal position of the vertical line between rule (i) and (i+1) is put in bound(i). In order to determine the bound(i) vector, a dynamic programming problem has to be solved, with as recursion formula :

$$\text{bound}(i) = \max_{\forall\, k=k^{*} \ldots k^{max}} \left[sl(k, tk(k,i)) + \text{bound}(i - a(k,i)) \;\middle|\; a(k,i) \leqslant a(k-1,i) \right] \qquad (1)$$

with $a(k,i)$ = number of rules immediately before i (included), having the same state for condition k

k^{*} = first condition in rule i which finishes its state.

The final table length is then given in bound(n) as this bound indicates the position of the last rule.

The restriction $a(k,i) \leqslant a(j,i), \forall\ j = 1 \ldots k$ inhibits constructions of the following type, which were found to be ambiguous :

Yes	No
-	

e.g. : without the restriction $a(k,i) \leqslant a(k-1,i)$ the example table would (partly) show like this :

Type of book		normal edition		pocket
Type of customer		-		
Quantity ordered		up to 100	more than 100	-
No discount	...	x	-	x
5 % discount		-	x	-
10 % discount		-	-	-

These constructions are in conflict with what could be called 'top down refinement', meaning that, moving from top to bottom, a box is never enlarged but always split up according to different condition states until one decision column is reached.
This process of 'top down refinement' facilitates the understanding and ease of decision making by human beings.

3.1.2. Assumptions

For ease of explanation, it is assumed at the moment that the screen is large enough to contain all the decision columns. In most cases however this is not true. We will account for the screen width in 3.1.5.

The length of a condition state cannot exceed a certain limit (which in the PRODEMO system is 20 characters), in order to be able to display at least one decision column on the screen.

We will also assume that the displayed length of the states is known. This is the number of screen positions occupied by a given state. Calculation of the displayed length means accounting for capital letters, accents and various special symbols as already mentioned in chapter 2.

No minimum is given to the width of the decision columns. In the actual PRODEMO algorithm, this minimum is set to 3 positions, because of visual attractiveness when one-symbol names are used, like Y, N, -, ...
The recursion formula would then use max(3,sl(k,tk(k,i))) instead of only sl(k,tk(k,i)), in order to obtain bound(i) - bound(i-1) $\geq$ 3.

3.1.3. Algorithm

In order to apply the recursion formula in (1), it is necessary to construct the a(k,i) matrix and to determine the critical condition k^*, i.e. the first condition in rule i which finishes a state.

The algorithm therefore can be split up in three parts :

- construction of the a(k,i) matrix
- determination of the critical conditions k^*
- calculation of the bound(i) vector

Though these three parts could be totally separated, the elements will be calculated rule per rule. There is no specific advantage in doing this, unless when the screen width is taken into account. Then the calculations depend on the number of rules which fit the screen.

The a(k,i) matrix indicates the number of rules immediately before rule i (included) which have the same state for condition k. As these states are the same, they should be displayed only once.

The critical condition of a certain rule is the first condition in that rule which terminates a state. Only the states of the critical to the last condition should be taken into account when calculating the required number of positions to display that rule. The other condition states continue in the following rule and will be considered when they are also finished.

The bound(i) vector is calculated by the formula which was given in (1). As the a(k,i) and k^* values are already known, this calculation is rather straightforward.

A formal solution to this text-editing problem is presented in the rest of this chapter.

The following notation is used :

n	: total number of rules
k^{max}	: number of conditions
$tk(k,i)$	: state number of condition k in rule i (with '-' = 0, irrelevant)
$s^{max}(k)$	: number of states of condition k
$sl(k,s)$	: length of state s of condition k
k^{*}	: first condition in rule i which terminates one of its states
$a(k,i)$	: number of condition states in previous rules up to i (included) which are equal to state tk(k,i) of condition k
$bound(i)$	: vertical position after displaying rule i, with bound(0) = stubwidth, bound(n) = total table width

with : $k = 1, \ldots, k^{max}$

$s = 0, \ldots, s^{max}(k)$
and $sl(k,0) = 1$ (i.e. '-')

$i = 1, \ldots, n$

A high level description of this algorithm and a generally coded form are supplied on the following pages.

high level description

for all rules i from 1 to n

(construction of the a(k,i) matrix)

```
a(0,i) := i
if i = 1 then a(k,i) := 1 , ∀ k
         else if a(k-1,i) > 1 and tk(k,i)=tk(k,i-1), ∀ k
                  then a(k,i) := a(k,i-1) + 1
                  else a(k,i) := 1
```

(determination of the critical conditions)

```
if i = n then critical condition k* := 1
         else k* := first condition with tk(k,i)≠tk(k,i+1)
```

(calculation of the bounds)

```
bound(i) := maximum  (state length + bound), cfr. (1)
            ∀k=k*..k^max
```

Note

a. (construction of the a(k,i) matrix)

As in the first rule all a(k-1,i) values are equal to 1, we can first test for a(k-1,i) > 1 and omit the i = 1 test. In our opinion this modification does not harm program readability or other similar objectives, and the result is a slight increase in efficiency. The i = 1 test is deleted in each rule, though the a(k-1,i) > 1 test is added to each condition of the first rule. As on average the number of rules is twice as high as the number of conditions, this improvement is entered in the final algorithm.

End note

The above high level algorithm can be refined as follows :

generally coded form

```
bound(0) := stubwidth
for all rules i := 1 to n
    (construction of the a(k,i) matrix)
     a(0,i) := i
     a(k^max,i) := 1
     for all conditions k := 1 to (k^max-1)
         if a(k-1,i)> 1
            then if tk(k,i) = tk(k,i-1)
                    then a(k,i) := a(k,i-1) + 1
                    else a(k,i) := 1
                 end if
            else a(k,i) := 1
         end if
     end for

    (determination of the critical conditions)
     if i = n
        then k* := 1
        else k := 1
             while tk(k,i) = tk(k,i+1)
                   k := k + 1
             end while
             k* := k
     end if

    (calculation of the bounds)
     bound(i) := stubwidth
     for all conditions k := k* to k^max
         bound(i) := maximum [bound(i),sl(k,tk(k,i))+bound(i-a(k,i))]
     end for
end for
```

Pass 1 : calculation of the rule boundaries

Notes :

a. construction of the a(k,i) matrix

- The test tk(k,i) = tk(k,i-1) should always be performed for the first condition except in the first rule as tk(k,0) is not defined. Therefore a(k-1,i), with k=1, should be > 1 if $i \neq 1$. This can simply be solved by assigning a(0,i) the value of i.
- The a(k,i) values of the last condition are always 1.

  proof : (ad absurdum) assume $a(k^{max},i) > 1$

  $\Rightarrow a(k^{max}-1,i) > 1$

  $tk(k^{max},i) = tk(k^{max},i-1)$

  $\Rightarrow a(k^{max}-2,i) > 1$

  $tk(k^{max}-1,i) = tk(k^{max}-1,i-1)$

  $tk(k^{max},i) = tk(k^{max},i-1)$

  $\Rightarrow tk(k,i) = tk(k,i-1), \forall\ k = 1 \ldots k^{max}$

  $\Rightarrow$ the two rules are identical (impossible)

  therefore : $a(k^{max},i) \leq 1$.
- The a(k,i) values of the first rule are always 1 as there is only 1 state per condition involved.
- The a(k,i) matrix therefore has the following fixed elements :

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \ldots & n \\ 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \vdots & \cdot & \cdot & \cdot & \cdot & \cdot & \vdots \\ 1 & 1 & 1 & 1 & 1 & \ldots & 1 \end{bmatrix} \qquad \begin{array}{l} k = 0 \ldots k^{max} \\ i = 1 \ldots n \end{array}$$

b. determination of the critical conditions

- The critical condition is the first condition which terminates a state in the given rule. In the last rule the first condition is always critical as all states are terminated. In the other rules it is the first condition with $tk(k,i) \neq tk(k,i+1)$.

loop invariant : $tk(j,i) = tk(j,i+1)$, $\forall\ j = 1 \ldots k-1$, $i \neq n$

proof of correctness :

```
k := 1
          ( ... loop invariant ... )
while tk(k,i) = tk(k,i+1)
          ( ... loop invariant ... ) ∧ (tk(k,1)=tk(k,i+1))
    k := k + 1
          ( ... loop invariant ... )
end while
          ( ... loop invariant ... ) ∧ (tk(k,i)≠tk(k,1+1))
```

proof of finiteness :

(ad absurdum) : assume $k > k^{max}$

$\Rightarrow$ (loop invariant) $tk(j,i) = tk(j,i+1)$, $\forall\ j = 1 \ldots k^{max}$

$\Rightarrow$ two identical rules (impossible)

therefore always : $k \leq k^{max}$

End notes

3.1.4. Illustrative example

The algorithm will be applied upon the example given in fig. 1 and fig. 2. The internal representation of the condition part for this table is :

1	1	1	2	2	3
1	1	2	0	0	0
1	2	0	1	2	0

with : n : number of decision rules = 6
k^{max} : number of conditions = 3

The length of the states has already been calculated :

hardcover : sl(1,1)=9
normal edition : sl(1,2)=14
pocket : sl(1,3)=6
wholesaler : sl(2,1)=10
retailer : sl(2,2)=8
up to 100 : sl(3,1)=9
more than 100 : sl(3,2)=13
- (irrelevant) : sl(1,0)=sl(2,0)=sl(3,0)=1

initialization

bound(0) = stubwidth = 20

rule 1

a(0,1) = 1
a(1,1) = 1
a(2,1) = 1
a(3,1) = 1

critical condition : $k^* = 3$
bound(1) = max (20,sl(3,1)+20) = 29

rule 2

$$\begin{aligned} a(0,2) &= 2 \\ a(1,2) &= 2 \\ a(2,2) &= 2 \\ a(3,2) &= 1 \end{aligned}$$

critical condition : $k^* = 2$

$k = 2$: bound(2) = max (20, sl(2,1)+20) = 30

$k = 3$: bound(2) = max (30, sl(3,2)+29) = $\boxed{42}$

rule 3

$$\begin{aligned} a(0,3) &= 3 \\ a(1,3) &= 3 \\ a(2,3) &= 1 \\ a(3,3) &= 1 \end{aligned}$$

critical condition : $k^* = 1$

$k = 1$: bound(3) = max (20, sl(1,1)+20) = 29

$k = 2$: bound(3) = max (29, sl(2,2)+bound(2)) = $\boxed{50}$

$k = 3$: bound(3) = max (50, sl(3,0)+bound(2)) = 50

rule 4

$$\begin{aligned} a(0,4) &= 4 \\ a(1,4) &= 1 \\ a(2,4) &= 1 \\ a(3,4) &= 1 \end{aligned}$$

critical condition : $k^* = 3$

$k = 3$: bound(4) = max (20, sl(3,1)+bound(3)) = $\boxed{59}$

Rule 5

a(0,5) = 5
a(1,5) = 2
a(2,5) = 2
a(3,5) = 1

critical condition $k^* = 1$

k = 1 : bound(5) = max (20, sl(1,2)+bound(3)) = 64
k = 2 : bound(5) = max (64, sl(2,0)+bound(3)) = 64
k = 3 : bound(5) = max (64, sl(3,2)+bound(4)) = 72

Rule 6

a(0,6) = 6
a(1,6) = 1
a(2,6) = 1
a(3,6) = 1

critical condition $k^* = 1$

k = 1 : bound(6) = max (20, sl(1,3)+bound(5)) = 78
k = 2 : bound(6) = max (78, sl(2,0)+bound(5)) = 78
k = 3 : bound(6) = max (78, sl(3,0)+bound(5)) = 78

The total table length is found to be 78 positions. It can easily be seen in fig. 1 that it is determined by the following states :

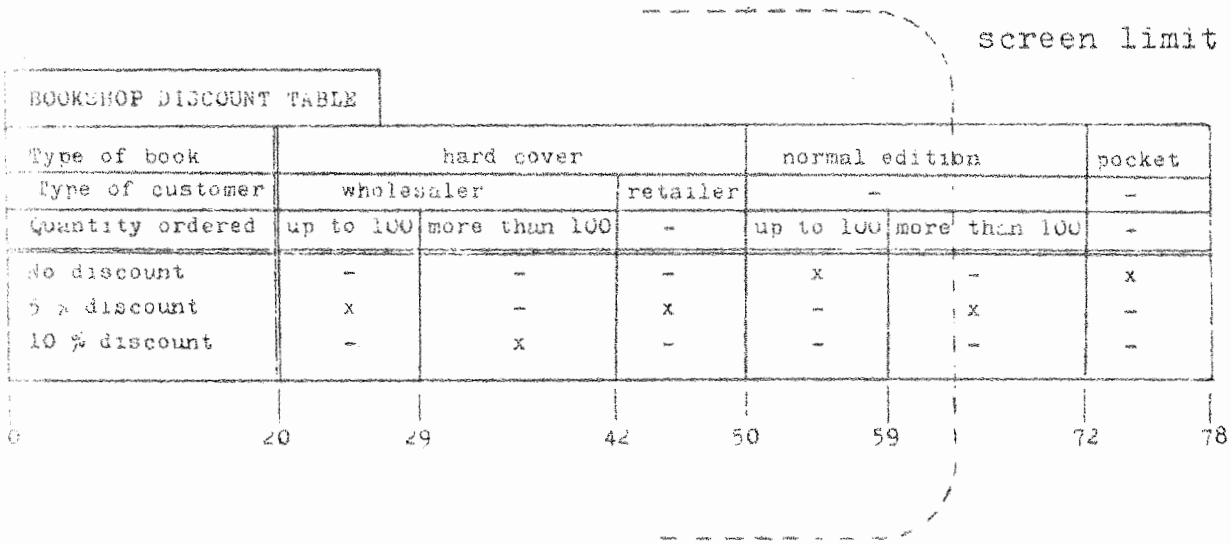
bound(0) = 20 = stubwidth
+ 9 = 'up to 100'
bound(1) = 29
+13 = 'more than 100'
bound(2) = 42
+ 8 = 'retailer'
bound(3) = 50
+ 9 = 'up to 100'
bound(4) = 59
+13 = 'more than 100'
bound(5) = 72
+ 6 = 'pocket'
bound(6) = 78 = total table length.

3.1.3. Adapting to the screen width

In most applications the screen width will not be sufficient to represent the complete decision table. The table then has to be cut after a certain rule, while the remaining rules are displayed on another page.

There are however a few problems in cutting the table. When a bound proves to be greater than the screen width, one can not simply take the next higher bound as all the states do not necessarily fit into this partial table.

Consider the following example :



After calculating bound(6)=72, it is seen that the screen limit is reached. It is not sufficient now to take the next higher bound (59) to plot the table, because 'normal edition' is longer than 'up to 100' and will not fit in these bounds, so that the screen limit may again be reached. The underlying reason is that the critical condition is no longer condition 3 but condition 1, as the end of the (partial) table is reached.

It will be clear by now that the procedure has to be adjusted in the following way : if a bound exceeds the screen width, take the previous bound and recalculate it with critical condition 1. If it still exceeds the screen width, take again the previous bound, etc...

This adjustment can also be implemented in the following way : Before calculating the normal bounds from the critical to the last condition, the bound from the first to the critical condition, called the noncritical bound, is determined in the same way (1). If at any time one of these two bounds is larger than the screen width, the table is displayed up to the previous rule.

The last form is choosen because the calculation of the noncritical bound is a fast and stable operation, while the backtracking of the first solution can produce unanticipated waiting times if it has to back up several rules.

The modified algorithm now has to be applied to every partial table until the last rule is reached. As the number of rules can be very high, the creation of a complete a(k,i) matrix and bound(i) vector would be a waste of memory. Therefore these values are relative to each partial table, so that the indices have to be adapted.

The final algorithm is roughly shown on the next page. The first two parts (determination of the critical condition and construction of the a(k,i) matrix) are similar to the previously given ones.

---

(1) Except for rules which can not exceed the screen width. If the previous bound is smaller than (screen width - maximal state length), the current state always fits into the screen width.

```
startrule := 1
while startrule ≤ n
    (start a new page and display the stub)
    bound(0) := totalbound := stubwidth
    rule := startrule - 1
    while totalbound < screenlength  and  rule < n
        rule := rule + 1
        i := rule - startrule + 1
        (construction of the a(k,i) matrix)
        (determination of the critical condition)
        (calculation of the bounds)
        if bound(i-1) ≥ screenlength - maximal range length
           then for k := 1 to k* - 1
                    calculate noncritical bound
                end for
                totalbound := maximum(noncritical bound,bound(i))
                if totalbound ≥ screenlength
                   then if totalbound > screenlength
                           then i := i - 1
                                rule := rule - 1
                        end if
                        if totalbound > screenlength  or
                                bound(i) < noncritical bound
                           then for k := 1 to k^max
                                    recalculate bound(i)
                                end for
                        end if
                end if
        end if
    end while
    (fill in the condition states and display the rules)
    startrule := rule + 1
end while
```

Pass 1, adapted to the screen width

3.2. Pass 2 : Putting the states in the rule boundaries

Once the positions of the vertical lines are known, we need to display the states within these boundaries, given the number of rules over which they extend. To this end we possess the following information :

- the decision table in internal representation (cfr. fig. 2)
- the number of rules n (= 6)
  the number of conditions k^{max} (= 3)
  the number of actions a^{max} (= 3)
- the length of the condition states
- the vector of rule boundaries
  bound(i) = [20, 29, 42, 50, 59, 72, 78]
- the a(k,i) matrix, k = 1, .. k^{max}

$$\begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 1 \\ 1 & 2 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

3.2.1. Methodology

As adjacent states have to be displayed only once, we need to know over how many rules a state extends. This information however can be found in the a(k,i) matrix :
from the definition :

$$a(k,i) = a(k,i-1) + 1 \overset{iff}{\Longleftrightarrow} tk(k,i)=tk(k,i-1) \wedge a(k-1,i) > 1$$

we know that

if $a(k,i) \geq a(k,i+1)$
then $tk(k,i) \neq tk(k,i+1)$ or $a(k-1,i+1) = 1$

so the state tk(k,i) is finished and can be displayed. The state extends over a(k,i) rules.

For each condition we can therefore examine the a(k,i) matrix, and display the given state if $a(k,i) \not> a(k,i+1)$. In this way a nice left-right, top-down display is obtained. There is however a little problem with the last rule. We can not test for $a(k,i) \not> a(k,i+1)$ as the latter is undefined. We would then have to display the last rule after the loop but according to the same procedure. In order to avoid this duplication in program code, a sentinel is added to each row of the a(k,i) matrix, i.e. an additional element which is assigned value 0. As a(k,n) is always $> a(k,n+1)=0$, the last rule is displayed in a nice and efficient way.

The display methodology now looks as follows :

- examine over how many rules a state extends
- look for the boundaries to plot between
- plot the state in the middle of these boundaries (1)

The foregoing only deals with the condition entries. The action entries cause no special problems as only a '-' or a 'x' has to be displayed. Notice that the action entries are displayed rule per rule which seemed slightly more efficient.

The following procedure is used to display the condition and action entries. No attention is paid however to the more technical details like : how and where to draw the lines, on which line should an entry be displayed, ...

3.2.2. Procedure

Bearing in mind the already mentioned absence of detail, the displaying procedure is rather straightforward :

---

(1) If there is an uneven number of positions available, PRODEMO uses half spaces to plot the states exactly in the middle of their bounds.

```
(condition entries)
for all conditions k := 1 to k^max
    a(k,n+1) := 0
    for all rules i := 1 to n
        if a(k,i) ⩾ a(k,i+1)
           then if tk(k,i) = 0
                   then between bound(i-a(k,i)) and bound(i)
                        write ('-')
                   else between bound(i-a(k,i)) and bound(i)
                        write (state tk(k,i) of k)
                end if
                draw vertical line after k at bound(i)
        end if
    end for
end for

(action entries)
for all rules i := 1 to n
    for all actions a := 1 to a^max
        if ta(a,i) = 0
           then between bound(i-1) and bound(i)
                write ('-')
           else between bound(i-1) and bound(i)
                write ('x')
        end if
    end for
    draw vertical line after all actions at bound(i)
end for
```

Pass 2 : displaying conditions and actions
between the rule boundaries

3.2.3. Final layout of the example problem

When accounted for a PLATO screen width of 64 positions, the example problem of fig. 1 would be displayed like this :

Bookshop discount table 04/22/80

Type of books	hard cover			normal edition
Type of client	wholesaler		retailer	-
Quantity ordered	up to 100	more than 100	-	up to 100
No discount	-	-	-	x
5 % discount	x	-	x	-
10 % discount	-	x	-	-

Type of books	normal edition	pocket
Type of client	-	-
Quantity ordered	more than 100	-
No discount	-	x
5 % discount	x	-
10 % discount	-	-

-back- to replot
-shift data- for menu -data- to touch END OF TABLE -shift back- to expand

4. CONCLUSION

The ideas expressed in this paper are the description of a small subsystem of the actual PRODEMO software.

An algorithm was presented to display decision tables of any kind, with the objective to minimize the required table length. In developing the algorithm, our main concern has been to produce efficient, reliable and readable software, according to structured programming objectives.

Actual results have proven to fulfill these objectives, with a very limited memory occupation and a quick response time.

5. ACKNOWLEDGEMENTS

The author wishes to thank PRODEMO pioneer Prof. M. VERHELST for his lively interest in this subject and for many valuable comments and suggestions. He is also very much indebted to R. MAES, who started the PRODEMO implementation on the CDC PLATO computer system and constructed its basic elements, many of which are reflected in this paper. The presented algorithm benefitted from his continuous cooperation and various interesting comments and advices.